AN OBJECT-ORIENTED SYSTEM FOR BAYESIAN NONLINEAR DESIGN USING XLISP-STAT

Merlise A. Clyde

School of Statistics, University of Minnesota

Technical Report #587

July 21, 1993

Abstract

An object–oriented programming environment in XLISP–STAT is developed for Bayesian nonlinear design. Prototypes and methods for Bayesian design objects are developed and examples from nonlinear regression, linear regression, mixture designs, and logistic regression are used to illustrate the methods.

1 Introduction

The goal of Bayesian optimal design is to find a design or a design measure on the design region that maximizes the expected utility for the particular experiment. This optimization problem involves several interrelated parts depending on the prior distribution, the design measure, the probability model for the response variable, and the utility function reflecting the purpose of the experiment. The object oriented environment in XLISP–STAT is ideal for developing software for computing Bayesian optimal designs. Models and distributions can be represented as objects with methods defined for computing design criteria, information matrices, or expectations of functions. New types of model objects, design criteria, and computational methods can be easily added without having to modify existing software. Graphical methods can also be used to check optimality of proposed designs. This paper describes objects and methods used in linear and nonlinear Bayesian design.

1.1 Problem Specification

In the design problem, the experimenter can select levels of the design variables x from a design region χ which is assumed to be a compact subset of $\mathbb{R}^k$. For each design point x_i, the experimenter independently observes the response variable Y_i. The response variable Y is assumed to be related to the design variables x and unknown parameters θ by a probability model with density $p(y|\theta, x)$.

Let Ξ denote the set of all probability measures defined on the design region χ. Then a generalized version of the design problem is to find an "approximate" optimal design by finding the optimal probability measure $\xi \in \Xi$. Exact n point designs can be obtained by restricting ξ so that $\xi(x_i) = 1/n$ for $i = 1, \ldots, n$. Further define $I(\theta, \xi)$ to be the normalized expected Fisher information matrix,

$$[I(\theta,\xi)]_{jk} = -\int E_{Y|\theta}[\frac{\partial^2}{\partial\theta_j\partial\theta_k}\log p(Y|\theta,x)]d\xi(x). \tag{1}$$

Many design criteria are functions of $I(\theta, \xi)$ and may approximate expected utility for some utility function. Most commonly used design criteria, denoted by $\phi(\xi)$, are real–valued concave functions of the design ξ. Finding approximate Bayesian optimal designs for nonlinear design problems using a design criterion ϕ generally involves numerical optimization of the form

$$\sup_{\xi\in\Xi} E_\theta[\phi(I(\theta,\xi))]$$

where the expectation is taken with respect to a prior distribution $\pi(\theta)$ defined on the parameter space $\Theta \in \mathbb{R}^p$ (Chaloner 1987). Additionally, the problem may require that some linear or nonlinear

constraints on the design measure that be satisfied. The next sections outline the object–oriented system for solving a Bayesian design problem. Several new prototypes for objects related to the design problem and methods for these objects are defined. It assumes that the reader is familiar with objects and methods in XLISP–STAT (Tierney, 1990) and examples are given that should aid in the development or modification of the methods.

2 Software

The optimization functions that are used to find optimal designs rely on the NPSOL FORTRAN library (Gill et al. 1986). To use the optimization routine, the C function must be dynamically loaded into XLISP–STAT or a static load of the routine must be done. On the DEC workstations in the School of Statistics at the University of Minnesota, a version of XLISP–STAT with the optimization code already incorporated is available. To use the XLISP–STAT version with the constrained optimization code, at the unix prompt enter

```
/ITASCA/users/clyde/bin/xlispstat
```

or

```
/ITASCA/users/clyde/bin/exls
```

to run `xlispstat` under `emacs`. The xlisp code for the Bayesian nonlinear design system must be loaded into XLISP–STAT. After starting up XLISP–STAT, enter

```
> (load "/ITASCA/users/clyde/thesis/lsp/bd-init.lsp")
```

at the XLISP-STAT prompt. This will load in all the prototypes and methods defined in this paper. Directions for static or dynamic loading of the optimization software is given in Section 5.

3 Bayesian Design Objects

A Bayesian design object is created based on the `bayesian-design-proto` prototype. This prototype contains slots for the different elements of the design problem: a probability model for the response variable Y, a prior distribution on θ, a design measure, the sample size, and the design region. Additionally there is also a slot for a title for the object. Methods can be defined for the `bayes-design-proto` for optimality criteria ϕ, finding optimal designs, and checking optimality of such designs. The various slots and the methods required by each of the corresponding objects defined for the `bayesian-design-proto` will first be described.

3.1 Probability Model

The `bayes-design-proto` has a slot for an object representing the underlying probability model. The only requirement for the `model` object is that it has a method that returns the information matrix as a function of θ and the design variables x. Some problems may include constraints that are functions that depend on the model. In this case, the `model` must have methods defined accordingly to return these functions.

Model objects for nonlinear regression can be based on the `nonlinreg-model-proto`, a prototype similar to the `nreg-proto` in LISP–STAT. As it is currently defined however, the `nreg-proto` does not work for design purposes since the mean function has to be a function of both θ and the design variables x.

The model object will be described using a one–parameter nonlinear model for the Rumford data set as described in Bates and Watts (1986, page 33). First a function for the mean has to be defined,

```
(defun rumford (theta x)
  (+ 60 (* 70 (exp (* x (- theta))))))).
```

For nonlinear regression models with independent normal errors the information matrix for a single observation x evaluated at θ is a function of the first derivative of the mean function ($\dot{\mu}(\theta, x) = (\frac{\partial \mu(\theta,x)}{\partial \theta_1}, \ldots, \frac{\partial \mu(\theta,x)}{\partial \theta_p})^T$), so that the Fisher information matrix is

$$I(\theta, \xi_x) = \dot{\mu}(\theta, x)\dot{\mu}(\theta, x)^T$$

where ξ_x is a design with all weight on the single point x. For the Rumford model, a function for the first derivative of the mean function is

```
(defun rumford1 (theta x)
  (* (- x)
     70
     (exp (* x (- theta)))))
```

The first derivative function could have been defined in terms of the mean function, `rumford`. Doing so, however, would cause problems with the `:save` method for the object.

In addition to the first derivative, the second and third derivatives are used for measures of non-normality and can be included in the model object,

```
(defun rumford2 (theta x)
  (* (^ x 2) 70 (exp (* x (- theta)))))

(defun rumford3 (theta x)
  (* (^ x 3) 70 (exp (* x (- theta))))).
```

These can be combined to create the `model` object using the function `make-model-object`,

```
(def rumford-model (make-model-object #'rumford
                                      #'rumford1
                                      :2nd-deriv #'rumford2
                                      :3rd-deriv #'rumford3
                                      :variance 441)).
```

The symbols for the mean function and the first derivative function are the only required arguments for `make-model-object`. When θ is a vector of parameters, the first derivative function must be the symbol of a function that returns a list or vector of the derivatives of the mean function with respect to θ. Similarly, the argument to `:2nd-derivative` should be a symbol for the function that computes the matrix of second derivatives of the mean function and the `:3rd-derivative` argument should be a symbol for a function that computes a 3–way array of the third derivatives. In the one dimensional case they can all be real–valued functions. The other arguments can be supplied or changed using accessor methods defined for the `nonlinreg-model-proto` prototype. For example, sending the nonlinear regression object the message `:mean` with no argument

`(send` ⟨*nonlinreg-object*⟩ `:mean )`

returns the symbol of the current mean function or, if the optional argument is present,

`(send` ⟨*nonlinreg-object*⟩ `:mean` ⟨*mean-function*⟩`)`

installs ⟨*mean-function*⟩ as the new value. Other accessor methods that are defined for the `nonlinreg-model-proto` are `:1st-deriv`, `:2nd-deriv`, `:3rd-deriv`, `:variance`, and `:title`, which all operate similarly.

As mentioned earlier, the model needs a method for returning the expected Fisher information matrix as a function of θ and x. For the `nonlinreg-model-proto` prototype, the method is defined as,

```
(defmeth nonlinreg-model-proto :inf-matrix (x theta)
  (let* ((df  (send self :1st-deriv) )
         (dfx (funcall df theta x)))
    (outer-product dfx dfx))).
```

Then

(send ⟨*nonlinreg-object*⟩ :inf-matrix ⟨x⟩ ⟨*theta*⟩)

returns the $p \times p$ information matrix evaluated at ⟨*theta*⟩ and ⟨x⟩.

Additionally the nonlinreg-model-proto has a :save method that allows the object to be saved to an ASCII file using the function save (a modification of the XLISP-STAT function savevar that allows functions to be saved in addition to variables and objects).

```
(def rumford-model-variable (send rumford-model :save))
```

creates a variable, rumford-model-variable, that can be saved to an ASCII file using the function save. Alternatively, one can use the save function directly since the save function will send the :save message to the object.

Prototypes and methods for other types of models such as generalized linear models can be defined similarly. The framework set up here will work for linear and nonlinear models with independent and identically distributed normal errors. Other models can use this prototype if the expected information matrix for a single design point x can be written in the form $w(\theta, x)g(\theta, x)g(\theta, x)^T$ where $w(.,.)$ is a real valued function. Then if the first derivative function for the nonlinreg-proto object is $(w(\theta, x))^{1/2}g(\theta, x)$, the correct information matrix will be returned using the existing methods. This approach is used to find optimal designs for a logistic regression model in section 4.2.

3.2 Prior Distribution

The prior distributions (as well as the design measures) in the system are represented as discrete distributions. The discrete-dist-proto is a simple prototype for discrete distributions. It has slots for probabilities, mass points, variable names, and a distribution name. This really is not a limitation, since quadrature or Monte Carlo methods of integration essentially are discrete approximations to integration. Continuous distributions can be represented by an appropriate discrete approximation, for example, a normal distribution could be represented using the points and weights used in a Gauss-Hermite quadrature method of integration.

The function make-dist-object can be used to create a new discrete distribution object,

```
(make-dist-object ⟨probabilities⟩
                  ⟨mass-pts⟩
                  :variable-names ⟨variable-names⟩
                  :dist-name ⟨name⟩)
```

where ⟨*probabilities*⟩ is a list of probabilities and ⟨*mass-pts*⟩ is a list of the support points of the distribution. For multivariate distributions, each support point would be represented as a list with ⟨*mass-pts*⟩ being composed as a list of these lists. The length of ⟨*probabilities*⟩ and ⟨*mass-pts*⟩ should be the same.

For example, to create a discrete bivariate distribution with three support points

```
> (def p (make-dist-object (list .25 .25 .5)
                           (list (list 1 1)
                                 (list 2 2)
                                 (list 0 10)))).
```

The two optional keyword arguments of `make-dist-object` are `:variable-names` and `:dist-name`. If there is only one variable, ⟨*variable-names*⟩ is just a character string, however for multivariate distributions ⟨*variable-names*⟩ should be a list of character strings. The ⟨*name*⟩ for the `:dist-name` keyword should be a character string. All of the above items can be obtained or changed using accessor methods for `:probabilities`, `:mass-pts`, `:variable-names`, and `:dist-name`. All return the current value if there is no optional argument, or reset the value if there is an optional argument. A related message, `:dist`, returns a list of the probabilities and mass points.

```
> (send p :probabilities)
(0.25 0.25 0.5)

> (send p :mass-pts)
((1 1) (2 2) (0 10))

> (send p :dist)
((0.25 0.25 0.5) ((1 1) (2 2) (0 10)))

> (send p :variable-names (list "x1" "x2"))
("x1" "x2")

> (send p :dist-name "2-dim discrete dist")
"2-dim discrete dist"
```

The message `:print` returns a formatted table for the distribution with variable names. This is also what appears if the name of the distribution object is entered.

Sometimes it is more convenient to define a distribution as the product of the marginal distributions when the variables are independent. The function `make-joint-dist` creates the joint distribution from a list of independent distributions. The function has one argument that is a list of distribution objects and returns a distribution object. Define a distribution for X as

```
> (def X (make-dist-object '(.25 .25 .25 .25) '( -1 -2 -3 -4)
                           :variable-names '("X1" "X2" "X3" "X4")))
X
```

and for Y as

```
> (def Y (make-dist-object '(.5 .25 .25) '((1 2) (3 4) (5 6))
                           :variable-names '("Y1" "Y2" "Y3")))
Y
```

then

```
> (def Z (make-joint-dist (list X Y)))
Z
> Z
A Discrete Distribution

p(X1 X2 X3 X4 Y1 Y2 Y3)     (X1 X2 X3 X4 Y1 Y2 Y3)

 0.125                     (-1 1 2)
 0.0625                     (-1 3 4)
 0.0625                     (-1 5 6)
 0.125                     (-2 1 2)
 0.0625                     (-2 3 4)
 0.0625                     (-2 5 6)
 0.125                     (-3 1 2)
 0.0625                     (-3 3 4)
 0.0625                     (-3 5 6)
 0.125                     (-4 1 2)
 0.0625                     (-4 3 4)
 0.0625                     (-4 5 6)
```

creates a distribution object Z for the joint distribution of X and Y. The joint distribution will contain the variable names from both distributions, if both distribution objects have variable names set.

The `discrete-dist-proto` has three methods for computing expectations. The first method is `:moments` which has an optional argument ⟨*power*⟩ of which the default value is 1. This method computes the expectations of powers of the discrete random variable. ⟨*power*⟩ can be either a scalar or a list of the same length as the number of variables in the discrete distribution object. Using `p` from the previous example,

```
> (send p :moments)
(0.75 5.75)
```

```
> (send p :moments '(2 1))
(1.25 5.75)
```

In addition to `:moments`, there are methods `:mean` and `:var-cov` which return the mean and the variance-covariance matrix, respectively, of the discrete distribution object. The second method is more general and is used to compute expectations of functions of the random variable and optional arguments by

`(send` ⟨*discrete-dist-object*⟩ `:fun-expectation` ⟨*fun*⟩ ⟨*args*⟩`)`

where ⟨*fun*⟩ is a function of the discrete random variable and optionally ⟨*args*⟩. For example, to find the expectation of $X1 * X2$ for the discrete distribution object p

```
> (defun cp (x) (* (first x) (second x)))
CP
> (send p :fun-expectation #'cp)
1.25
```

If the function has other arguments (which must follow the discrete random variable in the calling sequence of the function ⟨*fun*⟩) then these can be supplied as ⟨*args*⟩. For example,

```
> (defun g (x z) (* z (first x) (second x)))
G
> (send p :fun-expectation #'g 10)
12.5
```

where `z = 10`. The last method for computing expectations is similar to `:fun-expectation` but is used with other objects and their methods where the method has as arguments the discrete distribution random variable and optionally ⟨*args*⟩ as in `:fun-expectation`,

`(send` ⟨*discrete-dist-object*⟩ `:method-expectation` ⟨*object*⟩ ⟨*method*⟩ ⟨*args*⟩`)`.

This will used in section 3.3 and 3.7 to define a method for computing the normalized Fisher information matrix as in (1).

The `discrete-dist-proto` also has methods defined for finding the supremum and infimum of a function, ⟨*fun*⟩, over the support of the discrete random variable defined by the ⟨*discrete-dist-object*⟩. The usage is

`(send` ⟨*discrete-dist-object*⟩ `:sup` ⟨*fun*⟩ ⟨*args*⟩`)`

`(send` ⟨*discrete-dist-object*⟩ `:inf` ⟨*fun*⟩ ⟨*args*⟩`)`

where ⟨*fun*⟩ is a function of the discrete random variable and optionally ⟨*args*⟩.

For the Rumford model example, a two point prior distribution on θ,

```
(def prior  (make-dist-object  '(1 1) (list 1  10)))
```

can be used. Note that if the probabilities do not sum to 1, they will be normalized so that they do.

3.3 Design

The design objects are created from the `design-proto`. The `design-proto` inherits from the `discrete-dist-proto`. In this system, a design is represented as a discrete distribution by a list of probabilities and support points, and optional variable names and a design name. The `make-design-object` creates a new design based on this prototype.

For the Rumford example, the Bayes design object needs a starting design. A two point design can be used initially,

```
(def design0 (make-design-object  '(.5 .5) '(.1 1)))
```

in the Bayes design object.

Like the function `make-dist-object`, `make-design-object` has two keyword arguments, `variable-names` and `design-name`. The `design-proto` inherits all the methods of the `discrete-dist-proto` plus has some additional methods that pertain to designs and are used in the optimization procedures. The accessor method `:design-name` works like `:dist-name`.

The `design-proto` inherits the `:method-expectation` from the `discrete-dist-proto`. This can be used to compute the normalized Fisher information matrix as in (1).

```
(send design-object :method-expectation parammodel-object
                                        ':inf-matrix theta)
```

Remember that `:inf-matrix` had arguments `x` and `theta`, but with `:method-expectation` the expectation is taken with respect to the first argument of the method and the additional argument `theta` to the method `:inf-matrix` has to be supplied.

The above methods could be expanded to include parametric families of discrete and continuous distributions. Prototypes for the parametric families of distributions could inherit methods from the existing discrete distribution prototype, but a modification `make-dist object` could be defined that would instead create the parametric families using the parameters and family name.

3.4 Sample Size

The sample size n is represented as an integer. This is needed for exact designs or design criteria and constraints that include the sample size.

3.5 Support

The optimization methods for finding optimal designs use the support to constrain designs to be in the design region. For this method the design region is assumed to be the product of closed intervals. The support would then be represented as a list of lists in XLISP–STAT. The ith inner list would consist of the minimum and maximum value of the ith design variable for $i = 1, 2, \ldots, k$. For example, if χ is $[a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_p, b_p]$, then the support would be represented as,

```
(list (list a1 b1) (list a2 b2) (list a3 b3)  ...  (list ak bk))
```

Exchange type optimization algorithms (Fedorov 1972, Chapter 3) that search over a finite set of points could also be implemented, in which case the support would consist of a (finite) list of all possible fixed design points.

3.6 Creating a Bayes Design Object

In the previous sections all of the components of a Bayes design object were defined. They can now all be put together using the function `make-Bayes-design-object` to create a Bayes design object. The general syntax of the function is

```
(make-Bayes-design-object :model ⟨model⟩
                          :prior ⟨prior⟩
                          :design ⟨design⟩
                          :sample-size ⟨n⟩
                          :support ⟨support⟩
                          :title  ⟨title⟩)
```

The model keyword argument, ⟨*model*⟩, is a model object for the underlying probability model of the response variable. ⟨*prior*⟩ is a discrete distribution object and ⟨*design*⟩ is a design object. The sample size, ⟨n⟩, is an integer. The ⟨*support*⟩ is a list of lists where each inner list is the minimum and maximum of each design variable. ⟨*title*⟩ is a character string. For the Rumford model a Bayes design object named `rdo` is created from the `bayes-design-proto` prototype by

```
(setf rdo (make-Bayes-design-object
           :model rumford-model
```

```
        :prior prior
        :design design0
        :sample-size 2
        :support '((.00001 3))
        :title "rumford model"))
```

The bayes-design-proto has accessor methods for :model, :prior, :design, :sample-size, :support, and :title which can be used to return the current value if there is no optional argument or change the value if there is an optional argument. The bayes-design-proto also has a :save method so that Bayes design objects can be saved to ASCII files using the save function. In order to use the optimization methods, the :model, :prior, and :support keywords must be supplied. The other arguments can be added as needed.

3.7 Design Criteria

A design criterion for the nonlinear design problem will generally be a function of the normalized expected Fisher information matrix (1). A method for the bayes-design-proto that returns the normalized information matrix is defined as

```
(defmeth bayes-design-proto :norm-inf-matrix (theta design)
    (let ((model (send self :model)))
        (send design :method-expectation model ':inf-matrix theta)))
```

Chaloner (1987) used a criterion that approximates the Shannon information of an experiment,

$$\phi_1(\xi) = E_\theta[\log(\det(I(\theta,\xi)))]$$

The method that computes this is defined as,

```
(defmeth bayes-design-proto :phi1 (design)
  (let ((prior (send self :prior)))
    (send prior :fun-expectation
          #'(lambda (th d)
              (let ((detI (determinant
                           (send self :norm-inf-matrix th d))))
                (if (> detI 0) (log detI) *-INFINITY*)))
          design)))
```

Chaloner (1987) also used a criterion that approximates the negative weighted quadratic loss for estimating θ,

$$\phi_2(\xi) = -E_\theta[tr(B(\theta)I(\theta,\xi)^{-1})]$$

The method that computes this is defined as,

```
(defmeth bayes-design-proto :phi2 (design)
  (let* ((prior (send self :prior))
         (b-theta-fun (send self :b-theta))
)
    (- (send prior :fun-expectation
             #'(lambda (th d)
                 (let ((I (send self :norm-inf-matrix
                                th
                                d)))
                   (if (> (determinant I) 0)
                       (tr
                        (matmult
                         (funcall B-theta-fun th)
                         (inverse I)
                         ))
                     *INFINITY*)))
             design))))
```

The default value of `b-theta` is a function that returns the value one, however a particular function can be specified by

`(send` ⟨*bayes-design-object*⟩ `:b-theta` ⟨*fun*⟩`)`

where ⟨*fun*⟩ is function that returns the $p \times p$ matrix $B(\theta)$. Other criteria can be defined similarly by changing the lambda expression accordingly.

3.8 Optimization

Given a method for computing the design criterion, the message `:find-opt-design` can be used to find an optimal design measure,

`(send` ⟨*Bayes-design-object*⟩ `:find-opt-design` ⟨*criterion*⟩ ⟨*design*⟩`)`

The arguments are ⟨*criterion*⟩, a method that defines the optimality criterion, and ⟨*design*⟩ an initial design. The `npsol-max` function in XLISP–STAT is used to solve the optimization problem. The function `npsol-max` is used for maximization of a nonlinear function subject to nonlinear and linear constraints using routines in the FORTRAN library `npsol` (Gill et al. 1986). The constraints on the design region and constraints to ensure that the design measure is a valid probability measure are automatically taken into account. Additional linear and nonlinear constraints can be added using the keywords `:A` and `:constraints`, respectively. The argument for `:A` should be a matrix with dimensions l by k where each row of the matrix corresponds to a linear constraint of the

design variables and there are l linear constraints. Linear constraints on the design variables arise quite naturally with mixture designs. A list of nonlinear constraints on the design measure can be supplied as the argument to `constraints`. The nonlinear constraints should be implemented as methods for the ⟨*Bayes-design-object*⟩, and each constraint method should have as an argument a design object. Upper and lower bounds for the linear and nonlinear constraints can be supplied using `:lb-A`, `:ub-A`, `:lb-cx`, `:ub-cx`. If any of these are not supplied, then the bound is assumed to be a vector of $-\infty$ for the lower bounds `:lb-A` and `:lb-cx`, or ∞ for the upper bounds `:ub-A` and `:ub-cx`. The `:no-iterations` keyword controls the number of iterations. The default value is 500. The message `:find-opt-design` returns a design object.

The message `:find-exact-design` has the same arguments as `:find-opt-design` but is used to find an exact design in which case the optimization is only over the support points of the design. The probabilities of the starting design measure should be $1/n$ in order for the design measure to correspond to an exact design.

In many cases, particularly with exact designs, it is possible the algorithm will converge to a local maximum. Other starting designs can be used to see if there is any improvement in the criterion. Alternatively, algorithms such as simulated annealing (Haines 1987) that make minimal assumptions about the criteria could be used instead.

3.9 Directional Derivatives

For concave design criteria, the optimality of an approximate design can be checked by plotting the directional derivatives (Chaloner and Larntz 1989). If the design ξ^* is optimal, then the directional derivative from ξ^* in the direction of one point designs will be 0 at the support points of ξ^* and negative for other values in the design region. For problems with one or two design variables, the directional derivative can be plotted versus the design variables and used to check visually for optimality. Other types of graphical displays such as parallel plots or profile plots can be used to view directional derivatives when the design region is more than two dimensional. For each criterion, a method needs to be defined that computes the directional derivative. Then the `:plot-dir-deriv` message,

`(send` ⟨*bayes-design-object*⟩ `:plot-dir-deriv` ⟨*dir-deriv*⟩ ⟨*design*⟩`)`

plots the directional derivative defined by the method ⟨*dir-deriv*⟩ from the design ⟨*design*⟩ in the direction of one-point designs. For example, for the `:phi2` criterion, the directional derivative

method is implemented as

```
(defmeth bayes-design-proto :phi2-dir-deriv (opt-design x)
  (let* ((model (send self :model))
         (prior (send self :prior))
         (b-fun (send self :b-theta))
)
    (send prior
          :fun-expectation
          #'(lambda (th xpt)
                (let* ((x-design (make-design-object
                                  '(1) (list  xpt)))
                       (I-theta-x (send self :norm-inf-matrix
                                        th x-design ))
                       (I-theta-opt-inv (inverse (send self
                                                       :norm-inf-matrix
                                                       th opt-design )))
                       (b (funcall b-fun th)))
                  (+ (tr (matmult b
                                  I-theta-opt-inv
                                  I-theta-x
                                  I-theta-opt-inv))
                     (- (tr (matmult b I-theta-opt-inv))))))
          x)))
```

The directional derivative can be evaluated at any point in the design space, however the directional derivative currently will only be plotted if the design region χ is a subset of $\mathbb{R}^1$ or $\mathbb{R}^2$

4 Examples

4.1 Nonlinear Regression

For the Rumford example,

```
> (def phi1-opt-design (send rdo :find-opt-design ':phi1 optdes0 ))
PHI1-OPT-DESIGN
```

phi1-opt-design is the ϕ_1–optimal design object. To examine the design measure, the design can be sent the message :dist.

```
> (send phi1-opt-design :dist)
((0.536318 0.463682) (0.10649 0.999987)).
```

In this case the optimal design is a two point design. As the support of the prior distribution decreases, the Bayesian ϕ_1 optimal design becomes a design with only one support point at $1/E(\theta)$.

4.2 Logistic Regression

The logistic regression model corresponds to the problem of a Bernoulli response variable. Although the logistic regression model is not a nonlinear regression model with normal error structure, the methods for the `nonlinreg-proto` can still be used. Assume that for a single observation, the probability of success at the value x of the design variable is

$$p = P(Y = 1|\theta, x) = \frac{1}{1 + \exp -\beta(x - \mu)}$$

where β and μ are unknown parameters and $\theta = (\beta, \mu)^T$. Define,

$$\begin{aligned} w(\theta, x) &= (p(1-p))^{1/2} \\ \dot{\mu}(\theta, x) &= w(\theta, x)((x - \mu),\ -\beta)^T \end{aligned}$$

then the information matrix for this model for a single observation x evaluated at θ is

$$I(\theta, \xi_x) = \dot{\mu}(\theta, x)\dot{\mu}(\theta, x)^T.$$

This is an example of how, by using an appropriate function defined as $\dot{\mu}(\theta, x)$, the existing methods for the `nonlinreg-proto` will return the correct normalized information matrix. To define the model object, for example, it is only necessary for the optimal design functions to specify the correct function for the first derivative, however it is still necessary to give a function for the expected value of the response variable. To create the `logit-model`, define

```
(defun f1 (theta x) (* (select theta 0)
                       (- x (select theta 1))))

(defun f2 (theta x)
  (let* ((b (select theta 0))
         (u (select theta 1))
         (p (/ (+ 1 (exp (- (* b (- x u))))))
         (w (sqrt (* p (- 1 p)))))
    (* w (list (- x u) (- b)))))

(def logit-model (make-model-object #'f1 #'f2))
```

The function for `f1` is completely arbitrary.

The other parts of the Bayes-design object are defined as before. The prior distribution for β and μ is defined as

```
(def prior (make-dist-object '(1 1 1) (list (list 7 -1)
                                            (list 7 0 )
                                            (list 7 1 ))))
```

which corresponds to a independent prior distributions for β and μ, where the distribution for β has all its mass at 7 and the the distribution for μ has equally mass at -1, 0, and 1. Using a 2–point starting design,

```
(def design2 (make-design-object (list 1 1) (list -.5 .5)))
```

the Bayes-design object for this logistic regression problem is created by

```
(setf logitdo (make-Bayes-design-object :model logit-model :prior prior
                                        :design design2 :sample-size 2
                                        :support (list (list -1 1))
                                        :title "logit reg model")).
```

The 2–point ϕ_1 optimal design is found using the `:find-opt-design` method,

```
(def optdes2 (send logitdo :find-opt-design :phi1 design2))
```

and the directional derivative for the ϕ_1 criterion is plotted using

```
(def pp2 (send logitdo :plot-dir-deriv ':phi1-dir-deriv optdes2)).
```

This plot (Figure 1) indicates that a 2–point design is not optimal and so an equally spaced 7–point design was used as the starting design,

```
(def design7 (make-design-object (list 1 1 1 1 1 1 1)
                                 (list -1 -.5 -.25 0 .25 .5 1)))
(def optdes7 (send logitdo :find-opt-design :phi1 design7))

(def pp7 (send logitdo :plot-dir-deriv ':phi1-dir-deriv optdes7)).
```

The directional derivative (Figure 1) in this case does indicate that optdes7,

```
> optdes7
A Discrete Distribution

p(X0)                    (X0)

 0.171316                  -1
 0.177572              -0.593823
```

```
0.151113                -0.209335
   0                  6.87979e-07
0.151113                 0.209335
0.177572                 0.593823
0.171316                    1
```

is an optimal design. In this example, although a 7–point design was used as a starting design, there is positive mass on only six of the seven points in optdes7 returned by the `:find-opt-design` message.

4.3 Linear Models

Linear design problems from a non-Bayesian perspective (Silvey 1980) can also be solved using these methods. Consider the quadratic model,

$$y_i = \theta_0 + \theta_1 x_i + \theta_2 x_i^2 + \sigma\epsilon_i \tag{2}$$

where the ϵ's are independent and identically distributed normal random variables with mean 0 and variance 1. Assume that the design region χ is $[-1, 1]$. Then for this example the session would look like:

```
(def design0 (make-design-object '(.2 .5 .3) '(-.5 0 .5)))

(defun fun1 (theta x) (+ (select theta 0)
                         (* (select theta 1) x)
                         (* (select theta 2) (^ x 2))))

(defun fun2 (theta x) (list 1 x (^ x 2)))

(def lin-model (make-model-object #'fun1 #'fun2))

(def prior (make-dist-object '(1) (list (list 1 2 3))))

(setf lindo (make-Bayes-design-object :model lin-model :prior prior
                                      :design design0 :sample-size 3
                                      :support (list (list -1 1))
                                      :title "quadratic model"))

(def optdes1 (send lindo :find-opt-design :phi1 design0))
```

Note that even though the information matrix does not depend on θ, the methods assume that it does. For this reason, any prior distribution could be used and we use a one point prior distribution. In this situation, the Bayes ϕ_1 optimal design is the usual D-optimal design.

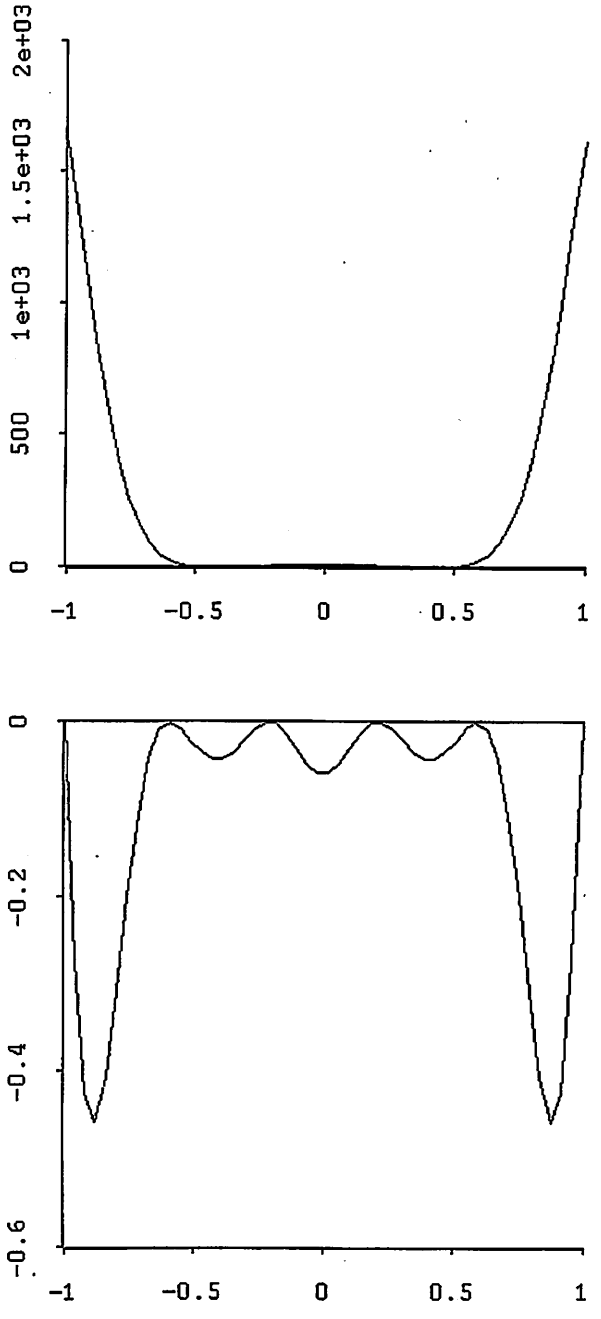

Figure 1: Directional derivative for the ϕ_1 two point optimal design and six point optimal design for the logistic regression model

Turning Point of a Quadratic Regression

If minimizing the asymptotic variance of the turning point of the quadratic, $-\theta_1/2\theta_2$, in the quadratic regression model (2) is of interest then the ϕ_2 criterion can be used (Chaloner 1989). First the function that defines the matrix $B(\theta)$ has to be defined and installed in `lindo`.

```
(defun b (theta)
  (let* ((theta1 (select theta 1))
         (theta2 (select theta 2))
         (ctheta (list 0
                       (- / 1 (* 2 theta2))
                       (/ theta1 (* 4 (^ theta2 2)))))
          )
        (outer-product ctheta ctheta)))

(send lindo :b-theta #'b)
```

Once the function is installed the `:phi2` criterion will use it. Sending the Bayes design object `lindo` the message `:find-opt-design` with the `:phi2` criterion and a starting design,

```
(def optdes2 (send lindo :find-opt-design ':phi2 design0))
```

will find an optimal design object, `optdes2`.

From the linear model example above, the directional derivative can be plotted by

```
(def ddp (send lindo :plot-dir-deriv ':phi2-dir-deriv optdes2))
```

Note that although the quadratic regression model is a linear model, this is nonlinear regression problem.

D–optimal Design for Two Design Variables

The previous problems have all involved only one design variable. Except for the plotting of directional derivatives all the methods work exactly the same for two design variables or more. Suppose the model of interest is the linear model with two design variables $X1$ and $X2$,

$$Y = \theta_0 + \theta_1 X_1 + \theta_2 X_2 + \sigma\epsilon. \tag{3}$$

The corresponding model object can be defined as

```
(defun fun1 (theta x) (+ (select theta 0)
                         (* (select theta 1) (select x 0))
                         (* (select theta 2) (select  x 1))))


(defun fun2 (theta x) (list 1 (select x 0) (select x 1)))

(def fun1-model (make-model-object #'fun1 #'fun2))
```

Using a 3-point starting design,

```
(def design3 (make-design-object (list 1 1 1 )
                                 (list '(-1 -1)
                                       '(0 0)
                                       '(1 -1))))
```

with support at three of the extreme points of the support, and a point mass prior distribution

```
(def pt (make-dist-object (list 1) (list (list 0 0 0))))
```

the Bayes design object is created by

```
(setf lindo (make-Bayes-design-object :model fun1-model :prior pt
                                      :design design3 :sample-size 4
                                      :support (list (list -1 1)
                                                     (list -1 1))
                                      :title "fun1 model")).
```

The methods to find optimal designs and plot directional derivatives for the ϕ_1 and ϕ_2 criteria are the same as before. Using the design3 as the starting design to find a ϕ_1 optimal 3–point design and checking the optimality using the directional derivative plot,

```
(def optdes3 (send lindo :find-opt-design :phi1 design3))

(def pp (send lindo :plot-dir-deriv ':phi1-dir-deriv optdes3))
```

it is clear that the 3–point design is not optimal (Figure 2). Repeating the above, but using a 4–point starting design, design4,

```
(def design4 (make-design-object (list 1 1 1 1)
                                 (list '(-1 -1)
                                       '(0 0)
                                       '(-1 1)
                                       '(1 -1))))

(def optdes4 (send lindo :find-opt-design :phi1 design4))

(def pp (send lindo :plot-dir-deriv ':phi1-dir-deriv optdes4))
```

the directional derivative indicates that the the design optdes4 is optimal (Figure 2). For two design variables, the directional derivative plot is based on spin-function, so the plot may be rotated to further examine the directional derivative. In the one design variable case there was a line where the directional derivative was zero, with two design variables there is a plane where the directional derivative is zero. The directional derivative should be equal to zero at the support points of the optimal design and below the zero plane everywhere else.

Constrained Mixture Design

Often the design variables correspond to the relative proportions of the components of a mixture. In addition to constraining the design points to the support of the design region, for each design point, the sum of the proportions must equal 1.0. There may be additional inequality constraints that have to be satisfied by the design points. DuMouchel and Jones (1992) describe an experiment with three design variables, A, B, and C. The model object for the mixture design is defined as

```
(defun mix-quad (theta x)
  (let ((xquad (combine x (^ x 2) )))
    (sum (* theta xquad))))

(defun mix-quad2 (theta x)
  (combine x  (^ x 2) ))

(def mix-quad-model (make-model-object #'mix-quad #'mix-quad2))
```

where the mean function is a quadratic model without the intercept term. For the ϕ_1 criterion, a degenerate prior distribution

```
(def prior-quad (make-dist-object '(1) (list (list 1 1 1 1 1 1))))
```

is used since the information matrix does not depend on θ. The ranges of the design variables are $0.10 \leq A \leq 0.75$, $0 \leq B \leq 0.4$, and $0 \leq C \leq 0.5$. This is used to define the support for the Bayes-design object,

```
(def mixsupport (list (list 0.1 .75)
                      (list 0 .40)
                      (list 0 .5)
                      )).
```

The starting design does not have to satisfy the constraints, although the rate of convergence will be affected. The starting design design0 is defined as,

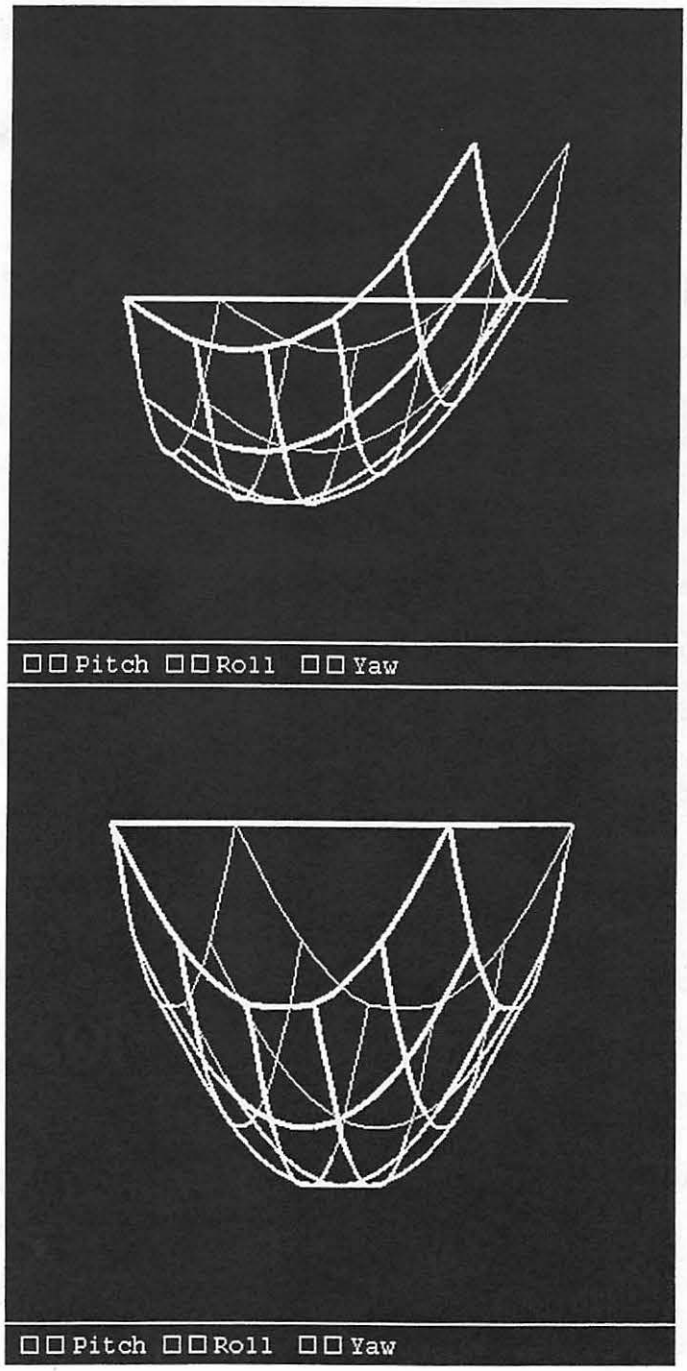

Figure 2: Directional derivative for the ϕ_1 three point optimal design and four point optimal designs for the linear model with two design variables.

```
(def design0 (make-design-object (repeat 1 9)
                                 (list '( .5 0 .5)
                                       '(.75 0 .25)
                                       '(.75 .25 0)
                                       '(.625 .375 0)
                                       '(.375 .375 .25)
                                       '(.125 .375 .5)
                                       '(.125 .375 .5)
                                       '(.3 .2 .5)
                                       '(.5 .2 .3))))
```

based on the design in DuMouchel and Jones (1992). The Bayes-design object is then defined as mixdo,

```
(setf mixdo (make-Bayes-design-object :model mix-quad-model
                                      :prior prior-quad
                                      :design design0
                                      :sample-size 9
                                      :support mixsupport
                                      :title "mixture model"))
```

In addition to the constraint that for a design point the design variables must sum to 1, ($A+B+C = 1$), DuMouchel and Jones (1992) specify an additional constraint that $0.5 \leq A + B \leq 1.0$. These two constraints can be specified as a matrix, a,

```
(def a (bind-rows  '(1 1 1)
                   '(1 1 0)))

(def lb-a (list 1.0 .5 ))

(def ub-a (list 1.0 1.0))
```

with lb-a and ub-a as the upper and lower bounds of the constraints in vector format. These are used as optional arguments to :find-opt-design to find the exact constrained D-optimal design,

```
(def optdes2 (send mixdo2 :find-exact-design :phi1 design0
                          :a a :lb-a lb-a :ub-a ub-a)).
```

The design object, optdes2, is the same as the design in DuMouchel and Jones (1992). Note because we are interested in finding the exact design, the choice of the starting design is very critical since the algorithm may converge to a local maximum. Other search algorithms or a grid of possible starting designs could be used.

Bayesian Design for Linear Models

Both the ϕ_1 and ϕ_2 criteria are based on an asymptotic normal approximation to the posterior distribution which ignores the prior precision matrix. In most instances, finding a design that minimizes the Bayes risk for a particular loss function is quite difficult, since this involves integration over both the prior and sampling distributions. An exception is the case of estimation for linear regression (Giovagnoli and Verdinelli 1983, Verdinelli 1983, Chaloner 1984, Pilz 1989). In particular, the case of estimation under quadratic loss quadratic loss or when interest in in a linear combination of the parameters (Chaloner 1984) will be illustrated. The usual normal linear model is assumed to be true,

$$Y = X\theta + \epsilon \tag{4}$$

where X is the n by k design matrix and ϵ conditional on σ has a n dimensional normal distribution with mean vector 0 and precision matrix wI where I is the $n \times n$ identity matrix. If the prior distribution for θ is also normal with prior mean θ_0 and prior precision matrix $w\tau$ where τ is a $k \times k$ known positive definite matrix, then the posterior distribution for θ given w is normal with mean $(\tau + X^TX)^{-1}(X^TY + \tau\theta_0)$ and precision matrix $(w(\tau + X^TX))$ (DeGroot pages 249-253, 1970). If we are interested in estimating θ under generalized quadratic loss, $(\theta - \hat{\theta})^TB(\theta - \hat{\theta})$, then the expected risk is $tr(B(\tau + X^TX)^{-1})E(w)$. To minimize the preposterior risk in estimating θ under this loss function, an approximate design should be chosen to maximize

$$-tr(B(\tau + nE_\xi xx^T)^{-1}) \tag{5}$$

where x is a $k \times 1$ vector of design variables. Note in this case the Bayesian optimal design will depend on the sample size n unlike the ϕ_1 or ϕ_2 optimal designs in the previous examples. This criterion is also appropriate in cases when the prior distribution is not normal (Tsutakawa 1972). The code necessary to define this criterion is similar to the ϕ_2 criterion, but we now have to include the prior precision. The ϕ_2 criterion could be viewed as a special case of this when the prior precision is 0.

Since the method `:precision` returns the prior precision (inverse of the variance–covariance matrix), the method for Bayes–A optimality can be defined as,

```
(defmeth bayes-design-proto :Bayes-A (design)
  (let* ((prior (send self :prior))
         (prior-precision (send prior :precision))
```

```
        (n (send self :sample-size))
        (b-theta-fun (send self :b-theta))
)
    (- (send prior :fun-expectation
             #'(lambda (th d)
                 (let ((I (send self :norm-inf-matrix
                                th
                                d)))
                   (if (> (determinant I) 0)
                       (tr
                        (matmult
                         (funcall B-theta-fun th)
                         (inverse  (+ prior-precision (* n I)))
                         ))
                     *INFINITY*)))
             design))
     )
)
```

The directional derivative is defined similarly as in the ϕ_2 criterion. Chaloner (1984) found designs for this criterion analytically, but noted that the solution sometimes gave negative weights for small sample sizes or for very precise prior distributions. Since the algorithm used in `:find-opt-design` to find the designs automatically constrains the weights to be non–negative, the designs may differ slightly from the analytic solution given in Chaloner.

5 Adding the Code for Constrained Optimization

The `npsol-max` function is based on a C function that calls the `npsol` FORTRAN library. The source code for this library is available from the

Office of Technology Licensing
350 Cambridge Avenue, Suite 250
Palo Alto, CA 94306

The C code for the interface must either be dynamically or staticly loaded in to XLISP–STAT. The file `optfront.c` contains the C code for the interface. It is necessary to specify the location of the header files for XLISP–STAT in order to compile `optfront.c` for dynamic loading. After compiling the file to `optfront.o`, it can be dynamically loaded into XLISP–STAT by using the expression

```
(dyn-load "optfront.o" :libflags "-lnpsol" :fortran t)
```

The :libflags argument may vary depending on where the library `npsol` is installed. If dynamic loading is not available or does not work, then the object code for `optfront.o` can be incorporated via a static load by adding `optfront.o` and the NPSOL library as extra objects and extra libraries in the makefile for xlispstat. This may be preferable, since the NPSOL library does contain print statements which may cause dynamic loading or the program to fail on some systems. The static loading has worked successfully on both DEC and NeXT workstations.

The source code for the C interface to NPSOL, and all xlispstat code used in this paper are available from the author. Please send a request to

`clyde@umnstat.stat.umn.edu`

or

`clyde@isds.duke.edu`

6 Conclusions

In this paper, several new prototypes and methods have been developed for Bayesian design. Through these tools, various nonlinear and linear design problem can be explored. Although, code written in xlisp must be interpreted, it is felt that interactive use may make it easier to explore effects of changing different components of the design problem. The software can be used to find approximate or exact designs, and locally optimal designs can be obtained by using a one–point (degenerate) prior distributions with the Bayesian criteria. Linear design problems such as the standard D–optimal criterion or A-optimal criterion are a special case where the design does not depend on the prior distribution. The Bayesian A-optimality criterion which does depend on the prior precision matrix is also obtained as a straight forward generalization. Mixture designs can be easily found using the constrained optimization. It is not necessary to have a starting design that satisfies the constraints. In addition to the linear constraints encountered in mixture design problems, nonlinear constraints (equality or inequality) on the design can also incorporated. Methods for constraints on parameter effects or intrinsic curvature for nonlinear models are available.

The methods and prototypes can be extended to handle other model types such as generalized linear models without changing the bayes-design-object-proto. Other classes of prior distributions can be developed based on the discrete distribution prototype. Other design criteria can be added by the user by defining new methods for the Bayes design prototype. Different optimization methods can be added without having to change existing code. For example, because there may be problems with converging to a local optimum, simulated annealing (Haines 1987) or similar algorithms could be used instead. The ability to use FORTRAN or C functions can also increase the efficiency of the program. A graphical user interface for describing design region and defining prior distributions as well as menus for models, selecting design criterion, choosing computational methods and directional derivative plots can be easily added.

References

Bates, D. M. and Watts, D. G. 1980. Relative curvature measures of nonlinearity. *J. Royal Statist. Soc. B*, **42**:1-25.

Chaloner, K. 1984. Optimal Bayesian experimental design for linear models. *Ann. Statist.* **12**:283-300.

Chaloner, K. 1987. An approach to design for generalized linear models. In *Proceedings of the workshop on Model-oriented Data Analysis, Wartburg* – Lecture Notes in Economics and Mathematical Systems, #297. Springer–Verlag, Berlin. 3–12.

Chaloner, K. 1989. Bayesian design for estimating the turning pont of a quadratic regression. *Communications in Statistics, Theory and Methods.* **18**:1385-1400.

Chaloner, K. and Larntz, K. 1989. Optimal Bayesian design applied to logistic regression experiments. *J. Statistical Planning and Inference*, **21**:191-208.

DeGroot, M. H. 1970. *Optimal Statistical Decisions.* McGraw–Hill Publishing Co., New York.

DuMouchel W. and B. Jones. 1992. A simple Bayesian modification of D–optimal designs to reduce dependence on an assumed model. Manuscript.

Fedorov, V. V. 1972. *The Theory of Optimal Experiments.* Studden, W. J. and Limko E. M. transl. and eds. Acedemic Press, New York.

Giovagnoli, A. and I. Verdinelli. 1983. Bayes D-optimal and E-optimal block designs. *Biometrika* **70**:695-706.

Gill, P. E., W. Murray, M. A. Saunders, and M. H. Wright. 1986. User's Guide for NPSOL (Version 4.0): a Fortran Package for Nonlinear Programming. Stanford Unifersity, Department of Operations Research, Technical Report SOL 86–2.

Haines, L. M. 1987. The application of the annealing algorithm to the construction of exact optimal designs for linear-regression models. *Technometrics* **29**:439-448.

Pilz, J. 1989. *Bayesian Estimation and Experimental Design in Linear Regression Models.* John Wiley & Sons, New York.

Silvey, S. D. 1980. *Optimal Design.* Chapman and Hall, New York.

Tierney, L. 1990. *LISP–STAT.* John Wiley & Sons, New York.

Tsutakawa, R. K. 1972. Design of an experiment for bioassay. *J. Am. Statist. Assoc.* **67**:584–590.

Verdinelli, I. 1983. Computing Bayes D- and A-optimal block designs for a two-way model. *The Statistician* **32**:161-167.